

Type-preserving compilation via dependently typed syntax in Agda

Andreas Abel¹

¹Department of Computer Science and Engineering
Chalmers and Gothenburg University, Sweden

Types for Proofs and Programs (TYPES 2020)
Torino, Italy
Scheduled for 2-5 March
(Cancelled due to COVID-19)

Verified Compilation

- Tony Hoare's Grand Challenge: Verified compilation.
- CompCert for the masses?
- Full verification may be too expensive ($> 90\%$ of impl. effort).
- Sweet spot: lots of confidence for little verification.
- Compiler be a **total** function.

Verifying Type-Safety

- Robin Milner: *Well-typed programs do not go wrong.*
- Types checked by compiler front-end.
- Goal: preserve properties through back-end.
 - Type safety.
 - “Execution safety”: No illegal jumps.
- Typed machine language (e.g. LLVM).

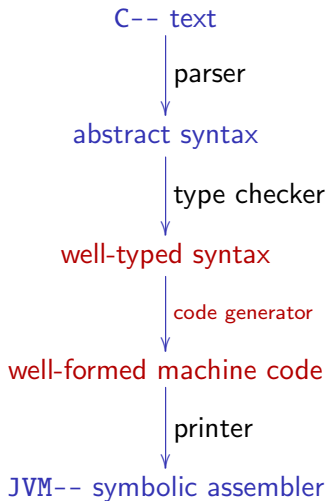
Method

- Implement compiler in a dependently-typed programming language.
- Represent well-typed syntax as indexed data types.
- Type-correct compilation enforced by indexing discipline.

Intrinsically well-typed syntax

object language	meta language
untyped e. g.: syntax trees	simply typed e. g.: (C, Java), Scala, ML, Haskell, ...
simply typed e. g.: λ -calculus, C--	dependently typed e. g.: Agda, Coq, Idris, Lean, ...
dependently typed	dependently typed

Pipeline



C-- by example

```
// Does p divide q?
bool divides (int p, int q) {
    return (q / p) * p == q;
}

// Is p prime?
bool prime (int p) {
    if (p <= 2) return p == 2;
    else {
        int q = 3;
        while (q * q <= p)
            if (divides(q,p)) return false;
            else q = q + 2;
    }
    return true;
}
```

C-- language elements

- Hierarchical:
 - function definitions contain statements,
 - statements contain expressions.
- Types: $Ty = \{int, double, bool, void\}$.
- Variables (function parameters, local variables) are *scoped*.
- Some statements declare new variables (`int q = 3;`).
- Control structures: `if`, `while`, `return`.

Typing contexts

- Scoping is managed by *typing contexts* Γ , snoc-lists of types.
- Example list $\text{int}^2 = [\text{int}, \text{int}]$:

$$\varepsilon.\text{int}.\text{int}$$

- Category Cxt :
 - Objects: typing contexts Γ .
 - Morphisms $\Gamma \subseteq \Delta$ are ways in which Γ is a sublist of Δ .

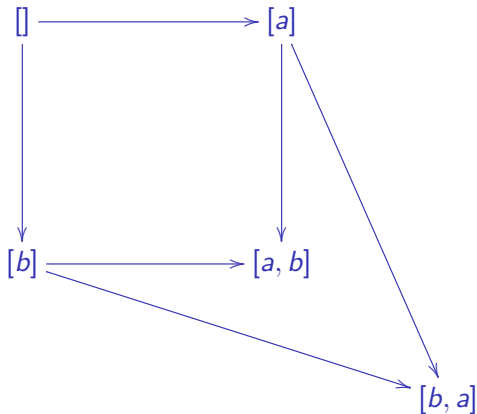
$$\text{skip} \frac{as \subseteq bs}{as \subseteq (bs.b)} \quad \text{keep} \frac{as \subseteq bs}{(as.a) \subseteq (bs.a)} \quad \text{done} \frac{}{\varepsilon \subseteq \varepsilon}$$

- Variables (de Bruijn indexes) pick a type from a context.

$$\text{Var}_t \Gamma \cong ([t] \subseteq \Gamma)$$

- Quiz:
 - How many morphisms in $\text{int}^2 \subseteq \text{int}^5$?
 - How many morphisms in $\text{int}^k \subseteq \text{int}^n$?

Cxt has only weak push-outs



Well-typed syntax

- $\text{Var}_t \Gamma$ variables of type t
- $\text{Exp}_t \Gamma$ expressions of type t
- $\text{Stm}_r \Gamma \Gamma'$ statements
 - r : return type of function
 - Γ : context before statement
 - $\Gamma' = \Gamma.\Delta$: context after
- $\text{Stms}_r \Gamma \Gamma'$ statement sequences: free category over Stm .

Expressions

- $\text{Exp}_t : \text{Cxt} \rightarrow \text{Set}$ functor
- maps hom $\eta : \Gamma \subseteq \Delta$ to weakening $[\eta] : \text{Exp}_t \Gamma \rightarrow \text{Exp}_t \Delta$
- constructors

lit	:	$(v : \text{Val}_t)$	$\rightarrow \text{Exp}_t \Gamma$
var	:	$(x : \text{Var}_t \Gamma)$	$\rightarrow \text{Exp}_t \Gamma$
arith	:	$(op : \text{ArithOp } t) (e_1 e_2 : \text{Exp}_t \Gamma)$	$\rightarrow \text{Exp}_t \Gamma$
cmp	:	$(op : \text{CmpOp } t) (e_1 e_2 : \text{Exp}_t \Gamma)$	$\rightarrow \text{Exp}_{\text{bool}} \Gamma$

Statements

$$\begin{array}{ll}
 \text{assign} : (x : \text{Var}_t \Gamma) (e : \text{Exp}_t \Gamma) & \rightarrow \text{Stm}_r \Gamma \Gamma \\
 \text{decl} : (t : \text{Ty}) & \rightarrow \text{Stm}_r \Gamma (\Gamma.t) \\
 \text{return} : (e : \text{Exp}_r \Gamma) & \rightarrow \text{Stm}_r \Gamma \Gamma \\
 \text{while} : (e : \text{Exp}_{\text{bool}} \Gamma) (s : \text{Stm}_r \Gamma \Gamma') & \rightarrow \text{Stm}_r \Gamma \Gamma \\
 \text{if} : (e : \text{Exp}_{\text{bool}} \Gamma) (s_1 : \text{Stm}_r \Gamma \Gamma_1) (s_2 : \text{Stm}_r \Gamma \Gamma_2) & \rightarrow \text{Stm}_r \Gamma \Gamma
 \end{array}$$

Java Virtual Machine (JVM)

- no registers
- stack for evaluating expressions
- local variable store (incl. function parameters)
- (heap for objects)
- method call handling behind the scenes

Java Virtual Machine (JVM) example

C--	Jasmin (symbolic JVM)
bool divides	.method divides(II)I
(int p, int q)	iload_1 ;; q
{	iload_0 ;; p
return	idiv
(q / p) * p == q;	iload_0 ;; p
}	imul
	iload_1 ;; q
	if_icmpeq L_true
	iconst_0 ;; false
	goto L_done
	L_true: iconst_1 ;; true
	L_done: ireturn
	.end method

Evaluation Stack

- JVM has local stack for evaluation of expressions.
- Stack type $ST = \text{List } Ty$
- Stack instruction $SI_{\Gamma} \Phi \Phi'$
 - $\Gamma : \text{Cxt}$ local variable store typing
 - $\Phi : ST$ stack typing before instruction
 - $\Phi' : ST$ stack typing after
- Constructors:

$$\begin{array}{lll}
 \text{ldc} & : (i : \text{Val}_{\text{int}}) & \rightarrow SI_{\Gamma} \Phi \quad (\Phi.\text{int}) \\
 \text{load} & : (x : \text{Var}_t \Gamma) & \rightarrow SI_{\Gamma} \Phi \quad (\Phi.t) \\
 \text{store} & : (x : \text{Var}_t \Gamma) & \rightarrow SI_{\Gamma} (\Phi.t) \quad \Phi \\
 \text{arith} & : (op : \text{ArithOp } t) & \rightarrow SI_{\Gamma} (\Phi.t.t) (\Phi.t)
 \end{array}$$

- Instruction sequences $SIS_{\Gamma} \Phi \Phi'$: free category over SI_{Γ} .

Variable typing administration

- Variable declarations $\text{decl } t : \text{Stm } \Gamma (\Gamma.t)$ are NOPs.
- Needed in intrinsically typed machine language.

$$\text{decl } t : (\Gamma, \Phi) \rightarrow (\Gamma.t, \Phi)$$

- Reconstruction in actual JVM by static analysis (bytecode verifier).
- Machine type $\text{MT} = \text{Cxt} \times \text{ST}$.

Jumps can go wrong

- Bad jump:

```

                                ;; Stack modification:
                                ;; [int,int] -> []
    if_icmpeq L_true             ;; [int,int] -> []
    iconst_0                     ;; []          -> [int]
L_true: istore_3                 ;; [int]       -> []

```

- Jump target needs to have same stack typing as source.
- Same for variable typing.
- Labels are typed by “before” machine type Ξ of target.
- Label context $\text{Labels} = \text{List MT}$.
- A label is a de Bruijn index $\ell : \text{Label} \equiv \Lambda$.

$$\text{Label} \equiv \Lambda \cong ([\Xi] \subseteq \Lambda)$$

Jump targets need to exist

- Semantics of a label is the code following it.
- Each label needs to point to some code.
- Two types of labels:
 - *Join points* for branches of `if` are `lets`.
 - *Back jumps* to repeat body of `while` are `fixs`.

Join points: let

```
[[ if (e) s1; else s2; s ]] =
```

```
let l = [[s]]  
    l1 = [[s1]]; goto l  
    l2 = [[s2]]; goto l  
in [[e]]; branch l1 l2
```

Back jumps: fix

```
[[ while (e) s0; s ]] =
```

```
let l2 = [[s]]
```

```
in fix l.
```

```
    let l1 = [[s0]]; goto l
```

```
    in [[e]]; branch l1 l2
```

Flowchart (control flow graph)

- $FC_r \Xi \Lambda$ control flow graph
 - r return type of method
 - Ξ machine state on entry
 - Λ typed jump targets
- Constructors:

exec	: $(i : SI_{\Gamma} \Phi \Phi')$	$(fc : FC_r (\Gamma, \Phi') \Lambda)$	$\rightarrow FC_r (\Gamma, \Phi)$	Λ
decl	: $(t : Ty)$	$(fc : FC_r (\Gamma.t, \varepsilon) \Lambda)$	$\rightarrow FC_r (\Gamma, \varepsilon)$	Λ
return	: $(e : Exp_r \Gamma)$		$\rightarrow FC_r (\Gamma, \varepsilon)$	Λ
goto	: $(\ell : Label_{\Xi} \Gamma)$		$\rightarrow FC_r \Xi$	Λ
branch	: $(o : CmpOp t)$	$(fc \ fc' : FC_r (\Gamma, \Phi) \Lambda)$	$\rightarrow FC_r (\Gamma, \Phi.t.t)$	Λ
let	: $(fc' : FC_r \Xi' \Lambda)$	$(fc : FC_r \Xi (\Lambda.\Xi'))$	$\rightarrow FC_r \Xi$	Λ
fix	:	$(fc : FC_r \Xi (\Lambda.\Xi))$	$\rightarrow FC_r \Xi$	Λ

Back end

- Code generation: translation from well-typed syntax to flow chart using continuations.
- Linearization: from flowcharts to basic blocks.
- Printing: from basic blocks to Jasmin symbolic JVM.

Evaluation

- *When it type-checks, it works.*
- Had only 3 bugs in compiler on first run!
- Agda programming requires hard thinking ahead.
- Little proof effort.
- Too hard for average beginning master student.
- Full verification in progress:
 - Needs reasoning in sublist-category.
 - Contributed categorical constructions (e.g. weak pushout) to Agda standard library.

Related Work

- Andrew Appell, Modern compiler implementation in C/Java/ML
- Xavier Leroy et al., CompCert, in Coq
- Magnus Myreen et al., CakeML, in HOL
- DeepSpec project: Verified tool chain
- Greg Morrisett et al., Typed Assembly Language
- Alberto Pardo, Emmanuel Gunter, Miguel Pagano, Marcos Viera, *An Internalist Approach to Correct-by-Construction Compilers*, PPDP'18: Terms indexed by semantics (in Agda)